National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2018/2019

Solutions for Recitation 10
Dynamic Programming & Memoization

Python

1. *Exception*

   Exceptions are error detected during execution. For example, `ZeroDivisionError`, `NameError` and `TypeError`

2. *Exception handling syntax*

```
try:
    statements
except <ErrorType>:
    statements
except (<ErrorType1>,<ErrorType2>, <ErrorType3>):
    statements
except:  # wildcard
    statements
finally: # always executed
    statements
```

3. *Optional else syntax*

```
try:
    statements
except <ErrorType>:
    statements
else:
    statements
```

4. Raising Exceptions

```
raise <ErrorType>
```

Problems

1. Recall the `count_change` algorithm shown in the earlier lectures. In a call to `cc(11, 5)`, there are actually computations of repeated subproblems, for example, repeated computation of `cc(1, 2)`.

```
# A certain path in the recursion tree:
cc(11, 5) -> cc(11, 4) -> cc(11, 3) -> cc(11, 2) -> cc(6, 2) -> cc(1, 2)
# Another path in the recursion tree:
cc(11, 5) -> cc(11, 4) -> cc(11, 3) -> cc(1, 3) -> cc(1, 2)
```

(a) Implement a memoized version of `cc(amount, kind_of_coins)` that uses a table to store previously calculated values.

```
def memo_cc(amount, kinds_of_coins):
  global count_memo
  count_memo += 1
  def helper(amount, kinds_of_coins):
    if amount == 0:
      return 1
    elif (amount < 0) or (kinds_of_coins == 0):
      return 0
    else:
      return memo_cc(amount, kinds_of_coins-1)
             + memo_cc(amount - first_denomination(kinds_of_coins),
                       kinds_of_coins)
  return memoize(helper, "cc")(amount, kinds_of_coins)
```

Alternative:

```
table={}
def memo_cc(amount, kinds_of_coins):
    if amount == 0:
      return 1
    elif (amount < 0) or (kinds_of_coins == 0):
      return 0
    else:
      if (amount, kinds_of_coins) in table:
         return table[(amount, kinds_of_coins)]
      else:
         ans = memo_cc(amount, kinds_of_coins-1)
             + memo_cc(amount - first_denomination(kinds_of_coins),
                       kinds_of_coins)
         table[(amount, kinds_of_coins)] = ans
         return ans
```

(b) Implement a dynamic programming version of `cc(amount, kind_of_coins)` that fills up a table systematically.

```
def dp_cc(amount, kinds_of_coins):
    row = [1]*(kinds_of_coins)
    table = []
    for i in range(amount+1):
        table.append(row.copy())

    for i in range(1,amount+1):
        for j in range(1,kinds_of_coins):
            d = first_denomination(kinds_of_coins-j+1)
            if i>=d:
                table[i][j] = table[i][j-1] + table[i-d][j]
            else:
                table[i][j] = table[i][j-1]

    return table[amount][kinds_of_coins-1]
```

2. Suppose you have a rod of length n meters and you hope to cut it into pieces and sell them. You can only cut them into integer lengths. The profit of rod with different lengths are displayed in the table below. How would you cut the rod to maximize the profit?

length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	30

Here is an example, suppose your rod is 4 meters. The different ways to cut rod and the corresponding profits are displayed below.

Do not cut at all. You make 9 dollars. Cut into 1 meter and 3 meters. You make $1+8=9$ dollars.
Cut into 2 meters and 2 meters. You make $5+5=10$ dollars. This is the optimal.
Cut into 1 meter, 1 meter and 2 meters. You make $1+1+5=7$ dollars.
Cut into four 1 meter rods. You make $1+1+1+1=4$ dollars.

Now write a function `cut_rod` that takes in an integer `n`, the length of the rod, and a dictionary mapping lengths to prices and returns the maximum profit that can be made. For example,

```
>>> prices = {1:1, 2:5, 3:8, 4:9, 5:10, 6:17, 7:17, 8:20, 9:24, 10:30}
>>> cut_rod(4, prices)
10
```

(a) Can you come up with a recursive solution?

```
def cut_rod(n, prices):
    if n<=0:
        return 0
    else:
        max_price = 0
        for piece in prices:
            if piece <= n:
                max_price = max(max_price,
                                prices[piece]+cut_rod(n-piece, prices))
        return max_price
```

Alternative:

```
prices = {1:1, 2:5, 3:8, 4:9, 5:10, 6:17, 7:17, 8:20, 9:24, 10:30}
prices = list(prices.values())
prices.sort()

def cut_rod(n, prices):
    if n<=0 or prices==[]:
        return 0
    else:
        d = prices[-1]
        if n>= len(prices): # sufficient length
            option1 = d + cut_rod(n-len(prices), prices)
            option2 = cut_rod(n, prices[:-1])
            return max(option1,option2)
        else:
            return cut_rod(n, prices[:-1])

print(cut_rod(4, prices) #10
```

(b) Can you give a solution that makes use of dynamic programming?

```
def dp_cut_rod(n, prices):
    row = [0]*(len(prices)+1)
    table = []
    for i in range(n+1):
        table.append(row.copy())

    for i in range(1,n+1):
        for j in range(1,len(prices)+1):
            if i>=j:
                table[i][j] = max(table[i][j-1],table[i-j][j]+prices[j])
            else:
                table[i][j] = table[i][j-1]
    return table[n][len(prices)]
```

Alternative:

```
prices = {1:1, 2:5, 3:8, 4:9, 5:10, 6:17, 7:17, 8:20, 9:24, 10:30}
prices = list(prices.values())
prices.sort()

def dp_cut_rod(n, prices):
    row = [0] * (len(prices)+1)
    table = []
    for i in range(n+1):
        table.append(row.copy())

    for i in range(1,n+1):
        for j in range(1,len(prices)+1):
            d = prices[j-1]
            if i>=j:
                option1 = d + table[i-j][j]
                option2 = table[i][j-1]
                table[i][j] = max(option1, option2)
            else:
                option2 = table[i][j-1]
                table[i][j] = option2
    return table[n][len(prices)]

print(dp_cut_rod(4, prices) #10
```